Article

My Bookmarks

-
-
-

Login or Register to enable bookmarks for unlimited time.

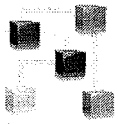
The content has been bookmarked!

There was an error bookmarking this content! Please retry.

Joshua Bloch: Bumper-Sticker API Design

Posted by Joshua Bloch on Sep 22, 2008
Community Java, .NET, Architecture, Ruby Topics Programming
Share |

Bookmark this!



My conference session *How to Design a Good API and Why it Matters* has always drawn large crowds; on InfoQ was the third most viewed content last year. When I presented this session as an invited talk at OOPSLA 2006, I was given the opportunity to write an abstract for the proceedings. In place of an ordinary abstract I decided to try something a bit unusual: I distilled the essence of the talk down to a modest collection of pithy maxims, in the spirit of Jon Bentley's classic Bumper-Sticker Computer Science, Item 6 in his excellent book, *More Programming Pearls: Confessions of a Coder* (Addison-Wesley, 1988).

Related Vendor Content

Twitter's JVM Performance Optimizations @ QConSF Nov 16-18

Scalable Agile: Building a Continuous Integration Pipeline with Go

Why NoSQL? A Primer on the Rise of NoSQL

Production Ready Software tutorial with Michael Nygaard @ QConSF Nov 16-18

Web API's: Design, Platforms, Performance – Track @QConSF Nov 16-18

Related Sponsor

QCon is the enterprise software development conference designed for team leads, architects, and project management. QCon runs five times a year in London, San Francisco, Tokyo, Beijing, and Sao Paulo. The next one is QCon San Francisco Nov 14-18, 2011.

QCon

It is my hope that these maxims provide a concise summary of the key points of API design, in easily digestible form:

All programmers are API designers. Good programs are modular, and intermodular boundaries define APIs. Good modules get reused.

APIs can be among your greatest assets or liabilities. Good APIs create long-term customers; bad ones create long-term support nightmares.

Public APIs, like diamonds, are forever. You have one chance to get it right so give it your best.

APIs should be easy to use and hard to misuse. It should be easy to do simple things; possible to do complex things; and impossible, or at least difficult, to do wrong things.

APIs should be self-documenting: It should rarely require documentation to read code written to a good API. In fact, it should rarely require documentation to write it.

When designing an API, first gather requirements—with a healthy degree of skepticism. People often provide solutions; it's your job to ferret out the underlying problems and find the best solutions.

Structure requirements as use-cases: they are the yardstick against which you'll measure your API.

Early drafts of APIs should be short, typically one page with class and method signatures and one-line descriptions. This makes it easy to restructure the API when you don't get it right the first time.

Code the use-cases against your API before you implement it, even before you specify it properly. This will save you from implementing, or even specifying, a fundamentally broken API.

Maintain the code for uses-cases as the API evolves. Not only will this protect you from rude surprises, but the resulting code will become the examples for the API, the basis for tutorials and tests.

Example code should be exemplary. If an API is used widely, its examples will be the archetypes for thousands of programs. Any mistakes will come back to haunt you a thousand fold.

You can't please everyone so aim to displease everyone equally. Most APIs are overconstrained.

Expect API-design mistakes due to failures of imagination. You can't reasonably hope to imagine everything that everyone will do with an API, or how it will interact with every other part of a system.

API design is not a solitary activity. Show your design to as many people as you can, and take their feedback seriously. Possibilities that elude your imagination may be clear to others.

Avoid fixed limits on input sizes. They limit usefulness and hasten obsolescence.

Names matter. Strive for intelligibility, consistency, and symmetry. Every API is a little language, and people must learn to read and write it. If you get an API right, code will read like prose.

Page 2 of 4

If it's hard to find good names, go back to the drawing board. Don't be afraid to split or merge an API, or embed it in a more general setting. If names start falling into place, you're on the right track.

When in doubt, leave it out. If there is a fundamental theorem of API design, this is it. It applies equally to functionality, classes, methods, and parameters. Every facet of an API should be as small as possible, but no smaller. You can always add things later, but you can't take them away. Minimizing conceptual weight is more important than class- or method-count.

Keep APIs free of implementations details. They confuse users and inhibit the flexibility to evolve. It isn't always obvious what's an implementation detail: **Be wary of overspecification.**

Minimize mutability. Immutable objects are simple, thread-safe, and freely sharable.

Documentation matters. No matter how good an API, it won't get used without good documentation. Document every exported API element: every class, method, field, and parameter.

Consider the performance consequences of API design decisions, but don't warp an API to achieve performance gains. Luckily, good APIs typically lend themselves to fast implementations.

When in Rome, do as the Romans do. APIs must coexist peacefully with the platform, so do what is customary. It is almost always wrong to â€œtransliterateâ€□ an API from one platform to another.

Minimize accessibility; when in doubt, make it private. This simplifies APIs and reduces coupling.

Subclass only if you can say with a straight face that every instance of the subclass is an instance of the superclass. Exposed classes should never subclass just to reuse implementation code.

Design and document for inheritance or else prohibit it. This documentation takes the form of selfuse patterns: how methods in a class use one another. Without it, safe subclassing is impossible.

Don't make the client do anything the library could do. Violating this rule leads to boilerplate code in the client, which is annoying and error-prone.

Obey the principle of least astonishment. Every method should do the least surprising thing it could, given its name. If a method doesn't do what users think it will, bugs will result.

Fail fast. The sooner you report a bug, the less damage it will do. Compile-time is best. If you must fail at run-time, do it as soon as possible.

Provide programmatic access to all data available in string form. Otherwise, programmers will be forced to parse strings, which is painful. Worse, the string forms will turn into de facto APIs.

Overload with care. If the behaviors of two methods differ, it's better to give them different names.

Use the right data type for the job. For example, don't use string if there is a more appropriate type.

Use consistent parameter ordering across methods. Otherwise, programmers will get it backwards.

Avoid long parameter lists, especially those with multiple consecutive parameters of the same type.

Avoid return values that demand exceptional processing. Clients will forget to write the specialcase code, leading to bugs. For example, return zero-length arrays or collections rather than nulls.

Throw exceptions only to indicate exceptional conditions. Otherwise, clients will be forced to use exceptions for normal flow control, leading to programs that are hard to read, buggy, or slow.

Throw unchecked exceptions unless clients can realistically recover from the failure.

API design is an art, not a science. Strive for beauty, and trust your gut. Do not adhere slavishly to the above heuristics, but violate them only infrequently and with good reason.

Watch Presentation: *How to Design a Good API & Why it Matters*

Joshua Bloch is Chief Java Architect at Google, author of Effective Java, Second Edition (Addison-Wesley, 2008), and coauthor of Java Puzzlers: Traps, Pitfalls, and Corner Cases (Addison-Wesley, 2005) and Java Concurrency in Practice. He was a Distinguished Engineer at Sun Microsystems, where he led the design and implementation of numerous Java platform features including JDK 5.0 language enhancements and the Java Collections Framework. He holds a Ph.D. from Carnegie-Mellon and a B.S from Columbia.

10 comments **Watch Thread** **Reply**

Documentation... by Seb Rose Posted 22/09/2008 06:59
- **Re: Documentation...** by Kurt Christensen Posted 22/09/2008 08:39
- **Re: Documentation...** by Albert Hwang Posted 22/09/2008 02:16

Converting use-case tests into examples in tutorials: code citing by Peter Arrenbrecht Posted 22/09/2008 11:06

I've read this before. by Jasper Novotny Posted 23/09/2008 03:17
- **Re: I've read this before.** by Abel Avram Posted 24/09/2008 01:05
  - **Re: I've read this before.** by Rich Unger Posted 26/09/2008 11:44
    - **Re: I've read this before.** by Ilja Preuß Posted 17/10/2008 09:06
- **Diamonds vs. stars** by Jaroslav Tulach Posted 25/09/2008 02:11

A collection of resources on API design by Christopher Bare Posted 25/07/2010 04:14

Sort by date descending

Documentation...

22/09/2008 06:59 by **Seb Rose**

"**APIs should be self-documenting:** It should rarely require documentation to read code written to a good API. In fact, it should rarely require documentation to write it."

"**Documentation matters.** No matter how good an API, it won't get used without good documentation. Document every exported API element: every class, method, field, and parameter."

A contradiction, I think.

Reply

Re: Documentation...

22/09/2008 08:39 by **Kurt Christensen**

There are always contradictory principles in engineering. That's why design is a non-trivial activity.

Reply

Re: Documentation...

22/09/2008 02:16 by **Albert Hwang**

The first refers to documentation of the code that uses the API. The second refers to the documentation of the API itself. If the API is properly named, then the code that uses it does not require documentation.

For example, if your API had a method that parsed a string and removed all white spaces, 'trim' would be much better than 'parse'.

```
// does not require documentation
a.trim();

// does require documentation
// parse string and remove all white spaces
a.parse();
```

Reply

Converting use-case tests into examples in tutorials: code citing

22/09/2008 11:06 by **Peter Arrenbrecht**

> Maintain the code for uses-cases as the API evolves. Not only will this protect you from rude surprises, but the resulting code will become the examples for the API, the basis for tutorials and tests.

I wrote a tool to cite snippets of Java source code into documentation for exactly this purpose. Supports omitting irrelevant detail and highlighting especially relevant parts.

You can see it in action in a largish project in that project's Quick Start example and all of the tutorial, as in this example.

Reply

I've read this before.

23/09/2008 03:17 by **Jasper Novotny**

These things are a lot in Jaroslav Tulach's new book. Only real difference is that 'diamonds' above are 'stars' there.

Reply

Re: I've read this before.

24/09/2008 01:05 by **Abel Avram**

> These things are a lot in Jaroslav Tulach's new book. Only real difference is that 'diamonds' above are 'stars' there.

Joshua Bloch's article is just a late follow-up of his presentation done in 2006 (www.infoq.com/presentations/effective-api-design). I hope you don't suspect Bloch plagiarizing Tulach.

Reply

Diamonds vs. stars

25/09/2008 02:11 by **Jaroslav Tulach**

There is a significant difference between diamonds and stars. While diamonds are said to be forever, nobody considers stars eternal. As such the allegories are not the same. They are in fact quite different. Read more...

Reply

Re: I've read this before.

26/09/2008 11:44 by **Rich Unger**

Whoa, I highly doubt anyone would ever suspect such a thing. Not least because Jaroslav's book is almost entirely devoted to disproving the last one: "API design is an art, not a science." He argues (convincingly, I think) that there are sound engineering principles which can be applied to design an API.

Reply

Re: I've read this before.

17/10/2008 09:06 by **Ilja Preuß**

> Jaroslav's book is almost entirely devoted to disproving the last one: "API design is an art, not a science." He argues (convincingly, I think) that there are sound engineering principles which can be applied to design an API.

So, are you saying that engineering is science, not art? ;)

Reply

A collection of resources on API design

25/07/2010 04:14 by **Christopher Bare**

I've collected a set of good resources on API design here:

digitheadslabnotebook.blogspot.com/2010/07/how-...

Reply